Replicated Architectures for Shared Window Systems: A Critique

*J. Chris Lauwers, Thomas A. Joseph, Keith A. Lantz and Allyn L. Romanow**

Olivetti Research California
2882 Sand Hill Road
Menlo Park, CA 94025

Abstract

Replicating applications in a shared window environment can significantly improve the performance of the resulting system. Compared to a completely centralized approach, a replicated architecture offers superior response time and reduces network load. To date, however, these advantages have been overshadowed by the equally significant synchronization problems associated with replication. In this paper we document these problems and show that the most frequent synchronization problems can be solved without changing existing software. We further indicate how some of the limitations of the resulting system can be removed by making applications or system servers collaboration-aware. Finally, we point out where general system support is needed to address the remaining deficiencies.

1 Introduction

A central aspect of computer support for group work is desktop teleconferencing. In the ideal, desktop teleconferencing enables geographically distributed individuals to collaborate, in real time, using multiple media (in particular, text, graphics, facsimile, audio, and video), from their respective offices. In addition, conferees are able to *share* applications, so that, for example, they can collaborate on the writing of a paper, the preparation of a budget, or the debugging of a program. One approach to application sharing is to develop special-purpose (*collaboration-aware*) applications designed for simultaneous use by multiple users (see, for example, [7, 17, 22, 26]). In many cases, however, it would be better if conferees could simply share existing single-user (*collaboration-transparent*) applications. In contemporary window-based environments, this requirement has led to the development of *shared window systems* [1, 5, 11, 13, 15, 20].

At the highest level of abstraction, a shared window system consists of a *conference agent* that is interposed between *applications* and the *window system* (Figure 1). The principal function of the conference agent is I/O multiplexing. Specifically, the conference agent distributes output streams from applications onto the users' window systems and merges the input streams from all users into a single input stream directed at the appropriate application(s). This activity enables participants in a teleconference to see the same view of every shared application and to interact with shared applications through their local input devices.[1] In contrast to terminal linking or shared-screen teleconferencing facilities [8], which require that users share everything on their screens, shared window systems permit a user to retain access to private applications during a teleconference. Shared window systems may also allow a user to take part in a number of teleconferences simultaneously, sharing different subsets of his applications (and associated windows) with different sets of users.

Most shared window systems have been implemented using variants of two basic architectures. Figure 2(a) presents the canonical *centralized* architecture: there is one instance of the conference agent and one instance of each shared application. All user input to a shared application is forwarded to the single instance of the application, and the application's output is sent to all the window systems for display. Figure 2(b) presents the *fully replicated* architecture: a copy of the conference agent and a copy of every shared application is executed locally on each user's

*E-mail: {lauwers,joseph,lantz,romanow}@orc.olivetti.com

[1] We refer to applications running under a shared window system as "shared applications." Similarly, windows associated with such applications are referred to as "shared windows."

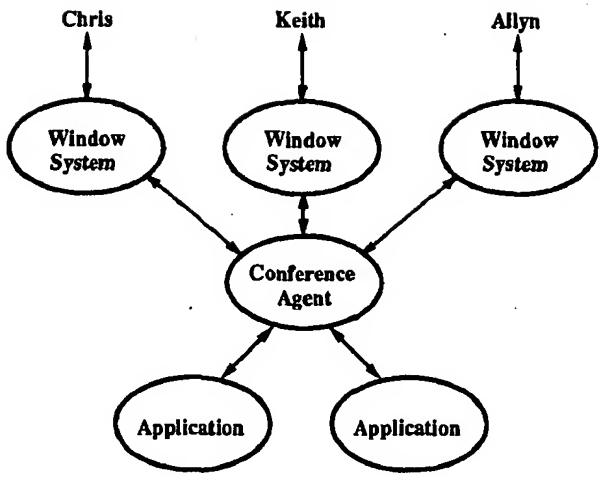

Figure 1: Abstract architecture of a shared window system.

workstation. User input to a shared application is distributed from the user's window system to all instances of that application. The output from each copy, however, is delivered only to the local window system. Although many variants of these basic architectures are possible, the remainder of this paper only deals with full application replication as in Figure 2(b).

Replication can significantly improve the performance of shared applications, as originally suggested by Sarin and Greif [21, 23]. Compared to a completely centralized approach, a replicated architecture offers superior response time and reduces network load. These benefits are sufficient to justify experimentation with replication and, indeed, motivated our work on VConf [15] and its successor, Dialogo [6]. The one other shared window system that employs application replication is MMConf, developed at BBN [5, 9]. The Colab project at Xerox PARC also adopted a replicated architecture for their collaboration-aware tools [10, 26].

Adopting a replicated architecture requires dealing with all the synchronization problems associated with replicated execution. Many solutions to these problems have been suggested [3, 4, 14, 25], all of which require extensive modifications to existing system software and applications. As a principal goal of shared window systems is to minimize modification to existing software, these approaches were originally deemed unacceptable. Rather, we decided to investigate how far we could push application replication without modifying any existing software.

The goal of this paper, then, is to document the synchronization problems associated with application replication in shared window systems. This discussion is based on our own experience with VConf and Dialogo, together with our understanding of the experiences of the MMConf team. We will show that the most frequent synchronization problems can be solved and that a usable system can be built without changing existing software. We will further indicate how some of the limitations of such a system can be removed by making applications or system servers collaboration-aware. Finally, we will point out where general system support is needed to address the remaining deficiencies.

The body of the paper is organized as follows. Section 2 reviews the motivations for adopting replicated architectures for shared window systems. Section 3 gives an overview of the challenges that application replication imposes on system designers. These challenges are discussed in detail in sections 4-6. Section 7 concludes with a summary of our approaches to the difficulties associated with replicated architectures.

2 Motivations for Application Replication

The main advantage of application replication is performance. Response time is better than in a centralized system because each user always receives output from a local copy of shared applications. Furthermore, depending on the window system, this output may not even have to go through the local conference agent, as was the case with VConf. This results in a further improvement in response time.

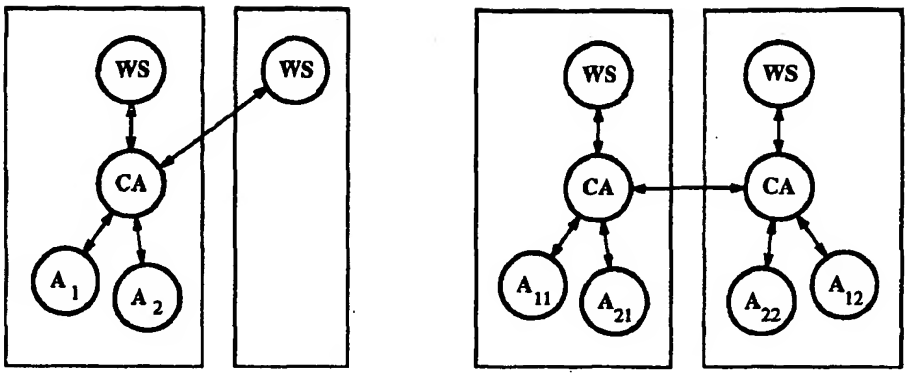

Figure 2: (a) Canonical centralized architecture. (b) Canonical fully replicated architecture. Circles should be interpreted as module instances—typically processes in separate address spaces. Boxes represent machine boundaries. The use of two subscripts in (b) indicates that the applications are replicated, with the first subscript identifying the application and the second the replica of that application.

Another performance benefit of application replication is that it typically lowers network traffic. In a centralized architecture, output data must be sent across the network to all window servers. When applications are replicated, each copy sends output only to the local window server, and no output data need be distributed across the network. The effect of this reduction in network traffic is magnified in typical workstation-based environments where the volume of output data sent to the window system dwarfs the volume of the input data received from it. Experiments performed on VConf and on Rapport [2] have verified these observations in the context of two different window systems—TheWA [16] and X [24], respectively. This reduction in bandwidth requirements suggests that a replicated architecture is particularly suited for teleconferencing across long-haul or slow networks.

A replicated architecture is also more versatile than a centralized one. In kernel-based window systems, such as SunWindows, the display hardware of a particular machine cannot be accessed by remote clients, so that a centralized architecture is not possible in such an environment. With kernel-based window systems, window sharing can only be achieved by adopting a replicated architecture.

More importantly, however, replicated architectures appear much better suited to accommodating heterogeneous hardware environments. Differences in hardware or other resources cause problems for any shared window system, whether based on a centralized or replicated architecture. In particular, X's imaging, color, and input models are neither consistent nor hardware-independent. Sizes and distances are measured in screen pixels, which vary from display to display. Keycodes are reported in vendor-specific terms. Applications can gain access to hardware-dependent information through objects called *visuals*, which abstract the properties of various displays. A companion paper [19] discusses this issue at length. In general, however, it should be easier to accommodate these differences using a replicated architecture because each instance of a shared application can (automatically) tailor itself to the display capabilities of its window server.

The same rationale applies when collaboration-aware applications must allow conference participants to use personalized interaction techniques or display different views on shared applications [17, 23, 26]. Again, these requirements are most easily handled in a replicated environment where each replica of a shared application can adapt its presentation to one participant's preferences.

3 Disadvantages of Application Replication: Overview

Application replication is not without its drawbacks. Indeed, some of these appear to be intractable under current technology. In particular, a replicated architecture will not work if an application to be shared is not available at all sites, or if it requires more resources (e.g. memory, devices, or licenses) than a particular participant's workstation can offer. Furthermore, it is difficult to add new participants to an ongoing teleconference, to move a participant

from one workstation to another, or, to provide support for spontaneous interactions in general. With a replicated architecture, these features require the ability to create up-to-date replicas of active applications on a new workstation. Thus, unless we have ubiquitous support for logging or process migration, these problems are likely to remain unsolved. These issues are discussed in detail in [18].

The more tractable problems of application replication arise mainly from the need to keep copies of shared applications synchronized with one another. If applications were to get out of synchronization, users would see different images on their screens and any further input would be interpreted differently on different machines. The synchronization problem is tractable when the shared applications are deterministic. An application is deterministic if, when given the same sequence of inputs starting from the same initial state, it always responds with the same sequence of outputs, and ends in the same final state. Also, output from a deterministic application does not depend on the timing between input events. In general, applications that are not deterministic cannot be handled in a replicated architecture. However, we shall see below that it is possible to control or tolerate certain forms of nondeterminism.

A related problem is the need to maintain *single-execution semantics* in the face of replicated execution. This means that executing multiple replicas must have the same effect on global system state as running a single replica.

How these problems manifest themselves depends greatly on the underlying window system. Since our discussion is based primarily on Dialogo, the solutions we present apply mainly in the context of X (Version 11). Nevertheless, the types of problems we describe are relevant to any replicated shared window system, and most of our solutions can be adapted to any underlying window system.

4 Input Consistency

The first step in keeping all replicas of a shared application synchronized is to ensure that they are started up in equivalent initial states and receive equivalent inputs. We say "equivalent" rather than "identical" because the inputs may have to be reinterpreted at each replica because of their different contexts. For example, window identifiers, sequence numbers and timestamps have to be remapped at each machine. This problem is complicated by the fact that an application may receive input from many sources besides the user. An application may read data from files, obtain the values of environment variables, communicate with the local window manager, receive messages from other applications, or access system servers. The conference agent guarantees that the same user input is delivered to all sites, but does not ensure the equivalence of input from other sources.

We can divide the problem of input equivalence into two parts: ensuring that applications receive the same (or equivalent) inputs from each individual source, and ensuring that inputs from the various sources are interleaved in the same order at all replicas. First, we describe how shared window systems guarantee equivalent user input. Then, we discuss how input from other sources can be handled. Finally, we discuss input ordering.

4.1 User Input

As discussed before, shared window systems use conference agents to distribute equivalent user input to all replicas of shared applications. Figure 3 shows the conference agent implementation for Dialogo, where conference agent functionality is distributed between a single conference manager and several conferee's agent processes, one per participant. Like VConf and MMConf, Dialogo uses the notion of a floor holder to regulate concurrent access to the shared workspace. Equivalence of user input is ensured by broadcasting input events generated by the floor holder to all replicas of conference applications. In the remainder of this paper, we refer to the floor holder's conferee's agent as the *active agent*, whereas all other conferee's agents are referred to as *passive agents*. Naturally, the floor holder changes as the conference progresses.

Generating consistent user input does not imply distributing the entire window system input stream to all participants, since not all events may be generated in response to user actions. X, for example, includes many events that are only used for *inter-client* communication. These events are generated by the X server in response to requests made by a client and inserted in the input stream of the intended receiver. For the remainder of this discussion, we classify X events in two classes:[2] user-input events (like the `ButtonPress` event), and inter-client events (like the

[2]For the discussion in this paper, events responsible for window refresh (like the `Expose` event) are not relevant and will not considered further.

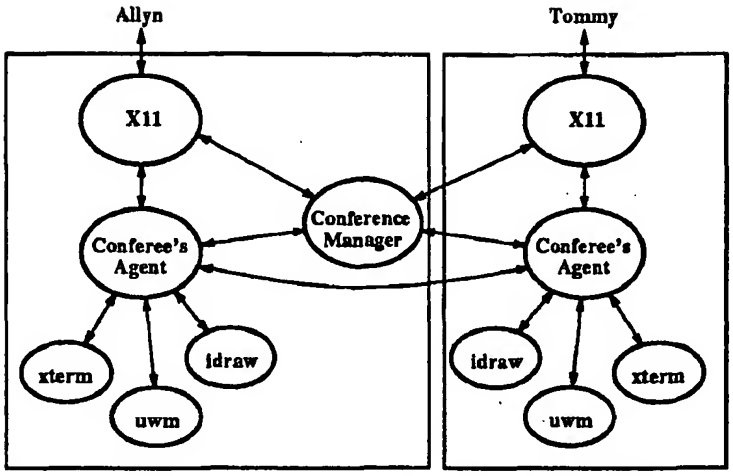

Figure 3: Architectural overview of Dialogo

MapRequest event). In general, a shared window system distributes user-input events only and does not broadcast inter-client events.

4.2 Input from Other Sources

There are two general approaches to ensuring that external sources generate the same input for all replicas. One is to replicate the external environment (or at least the parts of it used in a teleconference) and ensure that it remains the same at all sites. The other approach is to modify the external environment or its interface to make it *collaboration-aware*. In the latter approach, some of the entities in the external environment are modified so that they know when they are communicating with shared applications, and they generate consistent input to all replicas. However, one of the goals of a shared window system is to minimize modifications to existing software, so the solutions we describe below follow the first approach.

4.2.1 Files

At first glance, it may appear that the problem of reading input from files can be avoided by using a shared file system. However, if an application then *writes* to a file, each replica of the application would perform a write on the same file, and these multiple writes could lead to incorrect behavior (see Section 5). Dialogo, MMConf and VConf all address this problem by creating a separate "conference directory" for each participant. Dialogo and MMConf provide tools for creating and placing files in the conference directory. These tools ensure that any file placed in a user's conference directory is automatically copied into the other users' conference directories. In VConf, the user has to do this manually. As long as participants work within the conference directory and use relative path names to refer to files, they will all access identical copies of files.

Although this approach of copying working files into a conference directory is sufficient for most applications, it still has some shortcomings. If an application uses absolute path names to reference files, a different mechanism is needed to guarantee file consistency. Also, if a file copied into the conference directory contains a reference to another file, this reference may not be handled correctly. Under VConf and Dialogo it is up to the user to take care of these situations; MMConf circumvented some of these problems by adopting a collaboration-aware file system browser.

4.2.2 Environment Variables

Dialogo starts the first conference application (usually a window manager) with a standard set of environment variables, which will be inherited by all child applications. Participants in a teleconference may later change these values and add or delete environment variables. Since this action occurs on all workstations, the environment variables remain consistent. Dialogo also sets the value of the HOME environment variable to be the conference

directory and copies a standard set of user customization files into this directory, ensuring that shared applications will be customized the same way on all workstations.

4.2.3 Window Managers

Problems arise when shared windows are managed by each participant's private window manager. When a user resizes a shared window, for example, only his replica of the shared application will be notified of this change by his window manager, and the replicas of the application may become inconsistent. The general solution to this problem is to use collaboration-aware window managers, as discussed in [18]. These would coordinate between themselves to resize all instances of shared windows. Indeed, MMConf achieves this basic effect by broadcasting changes in window size to all conference agents. Since SunWindows allows clients to invoke window management functionality, these agents can adjust shared windows to keep their sizes consistent. However, under X, this becomes a problem when conferees are using different window management policies.

In contrast, Dialogo manages shared windows by running an existing window manager as a shared (replicated) application in a conference. Consequently, all communication between applications and the conference window manager is replicated on each participant's workstation. This ensures that the application and the window manager receive the same input and take the same actions on all workstations. One result of this approach is that Dialogo implements strict WYSIWIS workspaces—users have identical views of the shared workspace, including window placement and size.

4.2.4 Communication with Other Processes

Applications may also receive input from other applications and/or system servers. In fact, communication with the window manager, as described above, is just a special case of inter-process communication. In X and TheWA, clients communicate through the window server to implement functions like cut and paste. In MMConf, the folder browser application and the Diamond editor communicate directly when a file is opened through the browser. The way to ensure that the same input is received from other applications or system servers is the same as before—either replicate these processes or make them collaboration-aware. The more difficult problem, however, is to ensure that input from these sources is interleaved in the same order at all replicas. This is discussed next.

4.3 Ordering Input from Multiple Sources

Merely ensuring that all replicas of a shared application receive equivalent inputs from all sources is not sufficient, since events originating from different sources may be arbitrarily interleaved. Correct operation requires the entire input stream to be equivalent, meaning that all input events have to be delivered to replicated instances of shared applications in the same order.[3] The ordering problem usually does not occur for interactions with the file system, since applications typically use synchronous requests to read data from files. A problem occurs when multiple sources deliver input asynchronously. As an example, consider what happens under X when an application is started up. Applications typically create a new window and request that it be mapped, which results in a `MapRequest` event being sent to the window manager application. The window manager then waits for a `ButtonPress` event from the user to select the window position and size. Thus, the window manager is receiving input from both the application and the user. In a replicated environment, the `MapRequest` is generated in a replicated fashion on each participant's workstation, whereas the `ButtonPress` is generated by the floor holder and distributed to all replicas of the window manager. We have to ensure that the locally generated `MapRequest` and the floor holder's `ButtonPress` events are delivered in the same order to all replicas of the window manager.

Again, the general solution to this problem requires that applications be made collaboration-aware. In window-based systems, however, inter-process communication is often mediated through the window system. In both X and TheWA, for example, cut and paste operations are managed by the window server. Moreover, in X, all interactions between clients and the window manager application go through the server. The window server provides such *inter-client* communication by translating requests from one client into inter-client events sent to the destination client. These inter-client events are distributed over the same input stream as used for user-input events, and are therefore intercepted by the conference agent. This is an important observation, since it allows us to address ordering problems by implementing a synchronization algorithm in the conference agent.

[3] This requirement can be relaxed if application semantics are known [12]. We do not assume that shared window systems have knowledge about application semantics.

Dialogo takes the following approach (see [6] for a more detailed description). Since there is only one floor holder for the conference at any given time, Dialogo uses the event order on the active workstation as the unique ordering for all event packets. A naive implementation would distribute all event packets (inter-client events as well as user-input events) from the floor holder to passive agents, since this guarantees consistent ordering. However, this approach introduces another kind of synchronization problem: it could violate the causal relationship between the action that initiates an inter-client communication and the delivery of the corresponding inter-client event. In the example above, it was the application's call to `XMapWindow` that caused the delivery of the `MapRequest` event to the window manager. A passive agent could violate causality by delivering the floor holder's `MapRequest` event to the window manager before the application calls `XMapWindow` at that site. The window manager could then attempt to map the window even before it was created by the application, and would fail. If this happened on some sites but not on others, the system would get out of synchronization.

Therefore, Dialogo distributes both user-input and inter-client events from the floor holder to all passive agents, but the passive agents use the floor-holder's inter-client events only to impose an ordering relation on each participant's locally generated event stream. To achieve this ordering, passive agents queue locally generated inter-client events and deliver them to the application only as they are matched by corresponding inter-client events generated by the active agent. This guarantees that inter-client events are ordered consistently with respect to each other and with respect to user-input events. In the above example, then, this means that the passive conferee's agents queue the locally generated `MapRequest` event until the corresponding event is received from the floor holder, so that the correct order of delivery with respect to the `ButtonPress` can be determined.

Note that this solution is made possible by the existence of a single floor for the entire conference. If there is more then one floor per conference—as in MMConf, where there can be several *conversations*, each with its own floor—it is much more difficult to find a unique ordering relation for the event stream. The designers of MMConf have realized this, and require that communicating applications (like the folder browser and the editor) be part of the same conversation.

A final issue that needs to be addressed is the synchronization of input with respect to changing the floor. For example, if the current floor holder loses the floor while typing some input, it is important that all workstations agree whether that input occurred before or after the floor change. Dialogo employs a 2-phase protocol that guarantees that all participants agree on when a floor change occurs relative to the application input streams. Details of the algorithm may be found in [6].

4.4 Accommodating Non-Deterministic Input

As discussed earlier, applications that are non-deterministic cannot, in general, be run with a replicated architecture. However, disallowing all non-deterministic applications would be too drastic. A common source of non-determinism occurs when input queues are used by application programs to buffer events delivered by the window system. In particular, we refer to input queues implemented by the window system libraries, although this discussion applies to all queuing mechanisms that are not provided by the window server (pseudo ttys used by terminal emulators, for example, are input queues for terminal-oriented applications). Since most window-based applications are event-driven, the window system deposits events in these queues asynchronously, while at the same time the application processes queued events. This introduces the potential for race conditions between event delivery and queue manipulation operations, which results in non-deterministic behavior. Since the input queues and the routines that manipulate them are not implemented in the window server, a shared window system cannot prevent such race conditions. However, there are situations where these non-determinisms can be tolerated. This section discusses two common examples.

4.4.1 Polling the Input Queue

Many interactive applications contain code fragments that repeatedly execute the body of a loop until a particular input event arrives. After every iteration through the loop, the application program polls the input queue to check if the terminating event has arrived. Since there is no synchronization between the application polling the queue and the window system delivering the terminating event, the number of iterations through the loop is non-deterministic. The Diamond editor, for example, exhibits this behavior: holding down the mouse button in the scroll icon causes the editor to scroll through the document as long as the mouse button is depressed. In a computer conference, this mode of operation can cause replicated instances of the Diamond editor to scroll a variable number of pages. The

```
while(XCheckTypedEvent(display, ButtonPress, ...) == FALSE)
{
    XQueryPointer(display, rootWindow, ... );
    MoveOutline( ... );
}
```

Figure 4: A typical code fragment for rubber banding.

general solution to this problem is to modify all applications that exhibit this behavior. This is in fact the approach taken by the MMConf project.

However, if the body of the loop has no effect on application state, this form of non-determinism can be tolerated without modifying applications. A common example is the implementation of rubber-banding—the interactive display of an outline of an object being moved or resized. A typical code fragment for X is shown in figure 4; window managers often use code like this to draw an outline of a window as it is being moved. Although such code exhibits non-deterministic behavior, it can generally be tolerated since the body of the loop is only responsible for intermediate display actions and does not change application state.

In Dialogo, however, the code fragment causes the synchronization algorithm described in Section 4.3 to break. If different replicas of an application do not make exactly the same number of `QueryPointer` requests, different numbers of `QueryPointer` replies will be generated. This means that for some locally generated `QueryPointer` replies, there will be no matching reply coming from the floor holder. As a result, the locally generated replies will remain queued in the conferee's agents.

Therefore, to accommodate rubber banding, we slightly modified Dialogo's synchronization algorithm such that `QueryPointer` replies are not queued. Rather, whenever a `QueryPointer` reply is received from the floor holder, passive conferee's agents cache the latest information about the floor holder's pointer position. When a local `QueryPointer` reply is generated, pointer information in the reply packet is replaced by the cached information, and the reply passed on immediately to the local replica of the application.

4.4.2 Discarding Input Events

Another race condition occurs when application programs discard input events from the input queue. Since event delivery is asynchronous, input queues kept by different instances of shared applications may be different when the discard operation is performed, so that a different set of events will be discarded. The different replicas of the application may thus receive different input streams, and may take inconsistent actions as a result. A typical application that exhibits this behavior is `rn`, a Unix news reader program, which discards incoming events while its display is being updated. Using applications like `rn` in a computer conference will most likely result in inconsistencies.

Dialogo exhibits this problem because the `XSync` function provided by the X library. `XSync` allows an application to discard events that are pending in its input queue. For example, the `twm` window manager uses this feature to disable click-ahead when positioning a newly created window (presumably in order to prevent erroneous input). Since `XSync` is implemented completely in the window system libraries (there is no `XSync` request in the X protocol), a shared window system cannot prevent the resulting race conditions. On the other hand, if there were an explicit `XSync` request as part of the X protocol, the `XSync` reply could be used to delineate exactly the set of events that need to be flushed.

5 Output Consistency

Just as an application may receive input from many sources, it may also send output to various objects in the environment. For example, an application may write to a file, send a document to a printer, invoke a mailer, or turn on an actuator. Single-execution semantics need to be maintained when multiple copies of shared applications issue output requests to the same external object. For example, if the participants in a teleconference decide to send mail to someone, it is almost always the case that they intend only one copy of the mail to be sent. In contrast, if each replica of a shared application invoked a mailer, multiple copies would be sent. Moreover, if we replicate an external

object that receives output from many applications concurrently, these replicas might become inconsistent unless we ensure that output is delivered to each replica in the same order. (This is similar to the ordering requirement on input to shared applications.)

Fortunately, the applications typically used in a teleconference interact mainly with the window system and any external output is almost exclusively to the file system. Furthermore, these applications rarely write to the same files concurrently. In such situations output requests from different applications impose no ordering problems, and no concurrency control is required. The remaining problem is how to accommodate redundant output requests from replicas of the same application. The solution is to replicate the relevant files and have each application replica write to its own copies—that is, precisely the conference directory approach introduced in Section 4.2.1. These observations imply that a large class of applications can be used in a replicated system with no modification to system or application software.

However, if it is necessary to support concurrent access to the same file, then access to the file system must be regulated by a concurrency control mechanism that ensures that output operations are serialized in the same order on all copies of the file. Traditional replicated databases would *not* solve this problem. Such databases all assume that there is a single source for each write request, whereas here each write request is issued once from each copy of any application. One approach to solving this problem is to extend an existing replicated database to accommodate multiple instances of the same request. Another approach would be to modify the offending applications.

Similar observations hold with respect to objects other than files. That is, in the general case either the applications or the relevant object managers have to be modified to intercept redundant output requests and perform the operation only once. This approach was adopted by MMConf, which provides a collaboration-aware mailer and video information server, among other tools.

6 Start-up Consistency

The mechanism described in Section 4.3 synchronizes the input streams of applications that are already part of a teleconference. We also need to synchronize the starting up and introduction of new applications into a teleconference. If the workstations involved have different speeds, an application may start up on one workstation before it does on another. In this situation, the latter workstation may begin to receive events for an unknown application. To avoid this, Dialogo delays reading from or writing to a new application at the floor holder's site until all the passive agents have opened connections to replicas of the application.

A more difficult synchronization problem arises when applications are started up by a mechanism external to the conferencing system and there is no foolproof way for a server to identify new applications. With X on Unix, for example, a new application is started up when an existing application (usually a shell or a window manager) forks a new process—an action external to the conferencing system. The new application opens a connection to the conference agent using a well-known port. However, the conference agent has no way of identifying the application at the other end of a connection. This can lead to a problem when two (or more) applications are started at about same time (say by typing "xterm; emacs" to a shell). Each conference agent will receive two new connection requests, but there is no guarantee that the two applications will request connections in the same order at all workstations, and there is no simple way to tell which connection belongs to which application. If the conference agent makes the wrong choice, the input/output of one application at one site could be fed into the other application at another site, which would be disastrous.

Fortunately, this situation rarely occurs in practice. Participants in a teleconference typically spend most of their time browsing through shared windows and only occasionally start up new applications. They almost never start up multiple applications simultaneously. So, little damage is likely to result from leaving this problem unresolved, as we did in Dialogo.

For the sake of completeness, however, we indicate how the problem of start-up synchronization can be addressed. One general solution would be to somehow uniquely identify applications that open new connections. The conference agents could then use this information to synchronize application start-up. Unfortunately, it is not possible to uniquely identify applications without extensive support from the underlying system. While the application name or the command line arguments may suffice to differentiate between applications in most situations, a general solution should be able to disambiguate between multiple invocations of the same application, possibly with the same arguments. Such an approach would also require changing the underlying window system protocol.

No current protocol includes a mechanism for a new application to identify itself when it connects itself to a server. Hence, a solution based on identifying applications is impractical in the short term.

Another approach to start-up synchronization is to force the user to start up new applications only through the conference agent, as in MMConf, or through a conference-aware shell. The conference agent or shell then imposes the necessary synchronization. However, with this approach it is necessary to disallow users from themselves starting up any kind of shell application, lest they bypass the synchronization mechanism by starting applications from within their own shell.

7 Conclusions

Despite the potential benefits of replicated architectures for shared window systems, most groups have avoided them due to the significant synchronization problems that are associated with replicated execution. In this paper we have documented these problems, based primarily on our experience with VConf and Dialogo.

Keeping replicas of a shared application consistent requires that each replica receive (semantically) equivalent input from each input source and that input events originating from different sources be delivered to all replicas in the same order. The conference agent can guarantee equivalence for input generated by the base window system in response to user actions, but other mechanisms are required to guarantee input equivalence from other sources—file systems or window managers, for example. Two approaches are possible:

1. Replicate all relevant input sources.
2. Build collaboration-aware input sources, so that those sources can multicast their input to all replicas of shared applications.

The more difficult problem is to deliver input events originating from different sources in the same order to all replicas. In the most general case—where every source is generating input asynchronously, direct to the application—this requires that the application be made collaboration-aware.

As one of the principal goals of shared window systems is to minimize modifications to existing software, our intent with Dialogo was to investigate how far we could push application replication without changing existing software. Although conference agents can only guarantee equivalent user input, we have found that it is possible to replicate most other input sources—in particular the file system and the window manager. Moreover, in the case where the asynchronous input streams are all being mediated by the base window system (as is the case in a typical X environment), the conference agent can guarantee consistent input ordering. It is possible, therefore, to build a usable system without modifying existing software. Such a system, however, is not guaranteed to be robust when non-deterministic applications are used or when shared applications interact extensively with external entities that are not monitored by the shared window system.

To remove the constraints of such a basic system, it is necessary to make some system components collaboration-aware. This is in fact the approach taken by the MMConf team. Trivial modifications to applications or to system servers can often eliminate or prevent non-deterministic behavior. Building collaboration-aware window managers can remove the WYSIWIS-workspace constraint that results from replicating the window managers [18]. Collaboration-aware shells can prevent race conditions that arise when two conference applications are started simultaneously. Moreover, such shells can prevent users from running applications that are not robust in a replicated environment. Finally, single-execution semantics can often be preserved by modifying existing servers or by employing conference agent-like frontends to them.

Introducing collaboration-awareness into existing applications and/or servers can eliminate most of the problems associated with application replication. Solving the remaining problems—however rare—requires extensive additional support from the underlying operating system and from the available system servers. Input sources that cannot be replicated need to be modified and general system support for ordering events from arbitrary sources is required. Moreover, extensive system support is needed to support spontaneous interactions.

In general, we believe that the benefits of application replication are significant enough to warrant further attention, but that collaboration transparency cannot be maintained with current technology.

8 Acknowledgments

Thanks to Terry Crowley for his comments on an earlier version of this paper.

References

[1] S.R. Ahuja, J.R. Ensor, D.N. Horn, and S.E. Lucco. The Rapport multimedia conferencing system: A software overview. In *Proc. 2nd IEEE Conference on Computer Workstations*, pages 52–58. IEEE, March 1988.

[2] S.R. Ahuja, J.R. Ensor, and S.E. Lucco. A comparison of application sharing mechanisms in real-time desktop conferencing systems. Technical report, AT&T Bell Labs. To be presented at the COIS '90 Conference on Office Information Systems, ACM, April 1990.

[3] K.P. Birman, T.A. Joseph, T. Raeuchle, and A. El-Abbadi. Implementing fault-tolerant distributed objects. *IEEE Transactions on Software Engineering*, SE-11(6):502–508, June 1985.

[4] E.C. Cooper. Replicated distributed programs. In *Proc. 10th Symposium on Operating Systems Principles*, pages 63–78. ACM, December 1985.

[5] T. Crowley and H. Forsdick. MMConf: The Diamond multimedia conferencing system. In *Proc. Groupware Technology Workshop*. IFIP Working Group 8.4, August 1989.

[6] Desktop Teleconferencing Group. The Dialogo shared window system. Technical report, Olivetti Research California, in preparation.

[7] C. Ellis, S.J. Gibbs, and G.L. Rein. Design and use of a group editor. In *Proc. Working Conference on Engineering for Human-Computer Interaction*. IFIP Working Group 2.7, August 1989.

[8] D.C. Engelbart. NLS teleconferencing features: The Journal, and shared-screen telephoning. In *Proc. Fall COMPCON*, pages 173–176. IEEE, September 1975.

[9] H.C. Forsdick. Explorations in real-time multimedia conferencing. In *Proc. 2nd International Symposium on Computer Message Systems*, pages 299–315. IFIP, September 1985.

[10] G. Foster. *Collaborative Systems and Multi-User Interfaces*. PhD thesis, University of California - Berkeley, 1986.

[11] P. Gust. SharedX: X in a distributed group work environment. Presentation at the 2nd Annual X Conference, MIT, January 1988.

[12] M. Herlihy. *Replication Methods for Abstract Data Types*. PhD thesis, Massachusetts Institute of Technology, 1984. Published as Laboratory for Computer Science Technical Report MIT/LCS/TR-219.

[13] B. Janssen. Personal communication. February 1988.

[14] L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2(2):95–114, May 1978.

[15] K.A. Lantz. An experiment in integrated multimedia conferencing. In *Proc. CSCW 86 Conference on Computer-Supported Cooperative Work*, pages 267–275. MCC Software Technology Program, December 1986. Reprinted in I. Greif (editor), *Computer-Supported Cooperative Work: A Book of Readings*, pages 533-552. Morgan Kaufmann Publishers, 1988.

[16] K.A. Lantz. Multi-process structuring of user interface software. *Computer Graphics*, 21(2):124–130, April 1987. Presented at the SIGGRAPH Workshop on Software Tools for User Interface Development, ACM, November 1986.

[17] K.A. Lantz, J.C. Lauwers, B. Arons, C. Binding, P. Chen, J. Donahue, T.A. Joseph, R. Koo, A. Romanow, C. Schmandt, and W. Yamamoto. Collaboration technology research at Olivetti Research Center. In *Proc. Groupware Technology Workshop*. IFIP Working Group 8.4, August 1989.

[18] J.C. Lauwers and K.A. Lantz. Collaboration awareness in support of collaboration transparency: Requirements for the next generation of shared window systems. Technical report, Olivetti Research California. To be presented at the CHI '90 Conference on Human Factors in Computing Systems, ACM, April 1990.

[19] J.C. Lauwers and K.A. Lantz. Desktop teleconferencing and existing window systems: A poor match. Technical report, Olivetti Research California, in preparation.

[20] S. Piccardi and F. Tisato. Conference Desk: An experiment and model for application sharing. Technical report, Systems Software Laboratory, Direzione Olivetti Ricerca - Milan, January 1989.

[21] S.K. Sarin. *Interactive On-line Conferences.* PhD thesis, Massachusetts Institute of Technology, 1984. Published as Laboratory for Computer Science Technical Report TR-330.

[22] S.K. Sarin and I. Greif. Software for interactive on-line conferences. In *Proc. 2nd Conference on Office Information Systems*, pages 46–58. ACM, June 1984.

[23] S.K. Sarin and I. Greif. Computer-based real-time conferencing systems. *Computer*, 18(10):33–45, October 1985. Reprinted in I. Greif (editor), *Computer-Supported Cooperative Work: A Book of Readings*, pages 397-420. Morgan Kaufmann Publishers, 1988.

[24] R.W. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[25] R.D. Schlichting and F.B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[26] M. Stefik, G. Foster, D.G. Bobrow, K. Kahn, S. Lanning, and L. Suchman. Beyond the chalkboard: Computer support for collaboration and problem solving in meetings. *Communications of the ACM*, 30(1):32–47, January 1987. Reprinted in I. Greif (editor), *Computer-Supported Cooperative Work: A Book of Readings*, pages 335-366. Morgan Kaufmann Publishers, 1988.